AEROSPACE REPORT NO.
ATR-92(2778)-10

SDVS 1992 Final Report

30 September 1992

Prepared by

B. H. LEVY
Computer Systsems Division

DTIC
ELECTE
MAR 0 3 1995
S G D

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Engineering and Technology Group

19950227 056

THE AEROSPACE CORPORATION
El Segundo, California

PUBLIC RELEASE IS AUTHORIZED

DTIC QUALITY INSPECTED 1

Aerospace Report No.
ATR-92(2778)-10

SDVS 1992 FINAL REPORT

Accesion For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

Prepared by

B. H. Levy
Computer Systems Division

30 September 1992

Engineering and Technology Group
THE AEROSPACE CORPORATION
El Segundo, CA 90245-4691

Prepared for

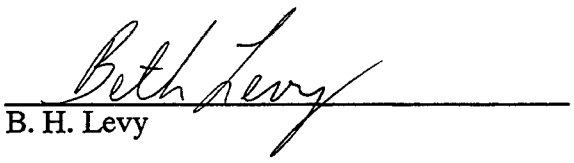
NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

PUBLIC RELEASE IS AUTHORIZED

Report No.
ATR-92(2778)-10

SDVS 1992 FINAL REPORT

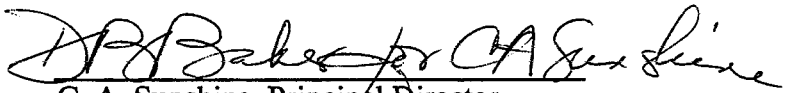
Prepared



B. H. Levy

Approved

D. B. Baker, Director
Trusted Computer Systems Department



C. A. Sunshine, Principal Director
Computer Science and Technology
Subdivision

Abstract

This report presents an overview of the progress made by The Aerospace Corporation on the State Delta Verification System (SDVS) project for the fiscal year 1992. The work documented in this report was funded by the National Security Agency through Air Force Space and Missile Systems Center contract number H98230-R292-9990.

Acknowledgments

The author gratefully acknowledges the contributions of the following people to the SDVS Verification Project and to the text of this report: Mark Bouler, Andrew Campbell, John Doner, Ivan Filippenko, Melodee Lydon, Leo Marcus, David Martin, Telis Menas, David Schulenburg, and Karl Schwamb.

Contents

Abstract v

Acknowledgments vi

1 Introduction 1

2 Software Verification Progress – Ada 5

2.1 Research and Development 5

2.2 Technical Reports 7

3 Hardware Verification Progress – VHDL 9

3.1 Research and Development 9

3.2 Technical Reports and Papers 12

4 Research in Safety and Pointers 13

4.1 Research 13

4.1.1 Safety 13

4.1.2 Ada Access Types (Pointers) 14

4.2 Technical Reports 14

5 General System Development 17

5.1 Development 17

5.2 Technical Reports 18

6 Conclusion 19

References 21

1 Introduction

This report presents an overview of the progress made in the State Delta Verification System (SDVS) project for the fiscal year 1992. The work documented in this report was funded by the National Security Agency through Air Force Space and Missile Systems Center contract number H98230-R292-9990.

The motivation for the SDVS project is the inadequacy of traditional large-scale software and hardware certification and analysis methods (e.g. testing or simulation) to ensure the correctness of computer systems. The goal of the project is to design and implement a system for an alternative computer-certification/analysis method called *verification*. Verification is the mathematical proof that a computer program or the design of a digital device satisfies a formal specification.

SDVS is the automated verification system being developed and used at The Aerospace Corporation. It aids in writing and checking correctness proofs. This project extends the capabilities and applicability of SDVS by incorporating various programming and hardware description languages, as well as by increasing the power of the verification system.

The long-term goal of this project is to create a production-quality verification system that can be used at all levels of the hierarchy of digital computer systems; the aim is to verify hardware from gate-level designs to high-level architecture, and to verify software from the microcode level to application programs written in high-level programming languages.

Figure 1 shows a typical hierarchy of a digital system. Each level in the diagram is implemented by the level below it. Until 1986 we focused on the verification of microcode. In the microcode application, the hardware description language ISPS was used to write computer specifications. At the microcode level, SDVS is used to prove that the microarchitecture executing some particular microcode implements a computer architecture correctly. Specifically, SDVS was used in the verification of the microcode implementation of the instruction set of the BBN (Bolt, Beranek and Newman) C/30 computer [1]. SDVS was successful in assisting us in finding errors in parts of the microcode and proving other parts of it correct.

We are currently extending the applicability of SDVS to both the lower levels of hardware design and the higher levels of computer programs. From 1987 until the present, we have

- added new programming and hardware description languages into the SDVS verification paradigm:
  - this included developing a new method for constructing translators from formal specifications of the languages' semantics, and
  - developing formal definitions of the domains specific to the computer application languages.
- increased the expressibility of SDVS by extending the underlying logic in several ways (e.g. by adding quantification, adding another temporal operator);

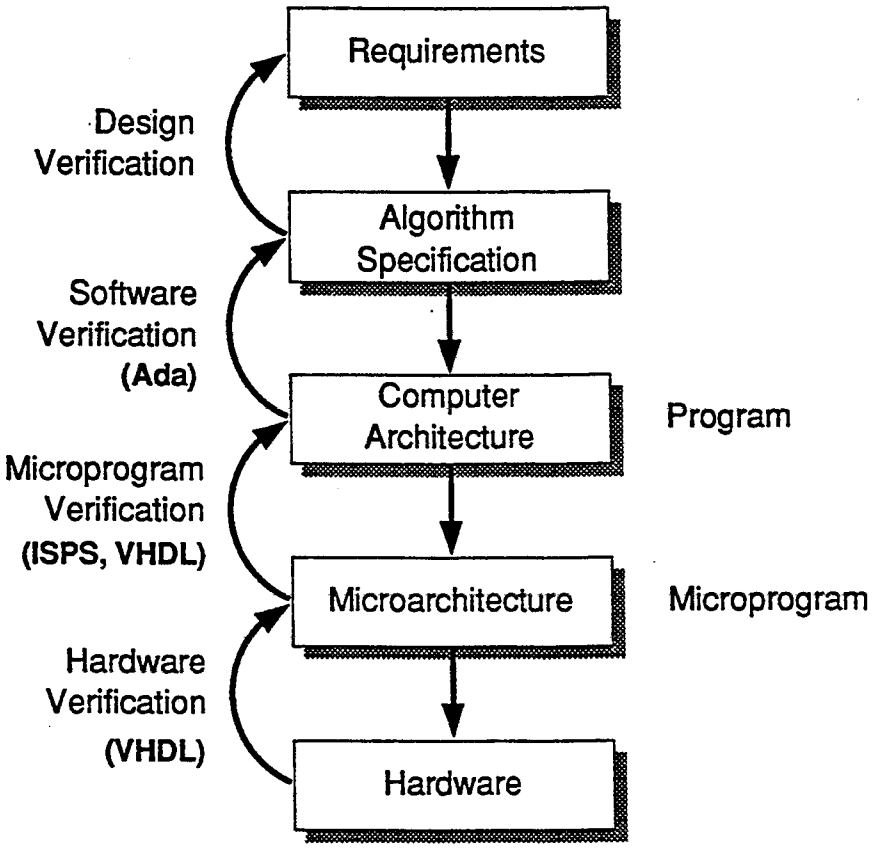


Figure 1: Verification Hierarchy

- developed a theory and extended the implementation for multilevel verification (verification at multiple hierarchical levels of computer systems);
- extended the system for specifying and reasoning about safety claims and concurrent computations; and
- studied the theory and designed the implementation for reasoning about recursive objects, permitting user-defined abstract data types, and making proofs more modular.

There are several features that characterize SDVS (a more detailed discussion is given in [2]). The underlying logic of SDVS is a temporal logic (called the *state delta logic*) that has a formal model-theoretic basis. SDVS has a sophisticated theorem prover that can be run in interactive or batch modes; the user supplies high-level proof commands, while many low-level proof commands are executed automatically. One of the more distinctive features of SDVS is its potential ability to incorporate widely used application languages (e.g. ISPS, VHDL, and Ada) and use descriptions in the application languages as either specifications or implementations in the verification hierarchy. Translators are used to translate application languages automatically to the language of the state delta logic.

In 1987 we developed a method for formally specifying translators and then systematically implementing translator specifications. A translator is specified by a set of mathematical equations that define the behavior of the translator; the equations define a denotational

semantics for the particular programming or description language. In 1988 we formally specified the Core-Ada-to-state-delta translator in terms of denotational semantics and then hand-coded this denotational description in Common Lisp. In 1989 we recognized the need for automating this process and developed, under an Air Force-funded project, a tool called DENOTE (Denotational Semantics Translator Environment) [3]. As part of this effort we defined a language called DL (DENOTE Language) for writing the equations. DENOTE translates specifications written in DL and outputs either formatted equations or a Common Lisp implementation of the translator. DENOTE was used this year to generate all of our translator implementations.

In addition to increasing our productivity, this method of implementing translators from their formal specifications has led to more reliable, better structured translators. The method also permits us to define incrementally the semantics of an application language to accommodate increasingly larger, more complex subsets of the language. This allows us to tackle the semantics of an application language without being overwhelmed by the size or complexity of the language. Incremental development also enables us to provide language subsets and achieve identifiable milestones.

The progress for this year is grouped into the following categories:

- adaptation and demonstration of SDVS to handle a subset of Ada
- adaptation and demonstration of SDVS to handle a subset of VHDL
- investigation of the semantics and proof techniques of certain language features common to Ada and VHDL (e.g. access types)
- specification and verification of safety properties using the extended state delta logic
- user support
- general upgrades of the system, including the rehosting of SDVS on the new version of Franz Allegro Common Lisp (FACL); SDVS can run on either Sun (Lucid) Common Lisp or FACL

Section 2 of this report gives a brief overview of this year's effort to adapt SDVS to handle a subset of Ada. This is part of a multiyear project to handle increasingly larger, more complex subsets of Ada, with the goal being to handle the entire language. In 1988 SDVS was extended so that programs written in an initial subset of Ada could be verified. In 1989-1991 three increasingly larger subsets were defined. Significant progress was made toward making the verification process more modular and eliminating unnecessary reverification; an extension to SDVS was implemented that permits one to prove properties of subprograms and then use those properties in proofs of programs that call the subprogram (referred to as *offline characterization*). Many small Ada programs (e.g. an Ada implementation of the quicksort algorithm) and procedures were verified. During the past four years we have focused primarily on enhancing the theory and system implementation of SDVS, to the point where we can now start to verify pieces of real applications.

In 1992, for the Ada effort, the main emphasis was on applications, although some modest enhancements were also made to our Ada capability. We feel that further development of the theory and automated tools must be driven by the consideration of real applications. Further effort is required to enhance SDVS for sizable applications, and the 1992 work described in Section 2 lists two such application efforts to help direct the development of the underlying theory and the implementation.

Section 3 gives a brief overview of this year's adaptation of SDVS to handle a subset of the hardware description language VHDL. A study into the feasibility of using SDVS for hardware verification started in 1988; this has led to our current work on extending SDVS so that hardware descriptions written in VHDL can be proved correct.[1] In 1989 the semantics and translator were developed for an initial subset of VHDL, and in 1990 enhancements were implemented that permitted correctness proofs of hardware descriptions written in this subset. In 1991-1992 the semantics and translator were developed for two increasingly larger subsets, and several VHDL hardware descriptions were verified.

Section 4 describes some additional research performed in support of the SDVS verification project. First, we have been studying how the logic can be used to prove safety properties and have further extended SDVS in order to reason about safety properties. Second, enhancements were defined and implemented in order to prove safety properties of terminating and nonterminating Ada programs. Third, we have been studying the semantics and proof rules for access types (pointers) in Ada. Work in all these areas is also applicable to VHDL.

Section 5 describes some general upgrades to the system and its documentation. A new version of SDVS, called SDVS 11, has been implemented this year. It has the enhanced Ada and VHDL verification capabilities. SDVS 11 runs on a Sun 4 computer (or a Sun 4-compatible computer, e.g. the Solbourne). Aerospace distributes all versions of SDVS, although the distribution must have prior approval by NSA. The current distribution policy is that SDVS can be distributed to universities and companies in the United States. It is still under export control, and organizations requesting SDVS outside the United States must undergo a separate request procedure.

This report is a brief overview of this year's major tasks that were funded by NSA. The reader is encouraged to review the technical reports and papers completed this year [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. Reference [16] is a related report published under a different project this year. Although [17] was published last year and thus is slightly dated, it still contains a good overview of the SDVS project.

[1] In our prior documents we have made an informal distinction between the verification of software, hardware, and microcode. In recent publications of research performed elsewhere we have seen microcode verification, a proof that a microarchitecture executing microcode correctly implements a computer instruction-set architecture, classified as hardware verification. Of course, this could also be called software verification. In this report microcode verification is distinguished from hardware and software verification.

2 Software Verification Progress – Ada

2.1 Research and Development

From 1988 to 1991 we adapted SDVS to handle four increasingly larger subsets of Ada, called Core Ada, Stage 1 Ada, Stage 2 Ada, and Stage 3 Ada. Core Ada has basic statements (e.g. assignments, conditionals, and loops), simple input/output, packages, and basic data types (integer, boolean, and array). Stage 1 Ada is a superset of Core Ada that adds subprograms, records, enumeration types, and type declarations to Core Ada. Stage 2 Ada adds user-defined exception handling and the character data type to Stage 1 Ada. Stage 3 adds the string data type and a standard package mechanism with a portion of the standard `TEXT_IO` package to Stage 2 Ada. Stage 3 Ada is roughly comparable to Pascal; it does not have Pascal's fixed/floating point numbers, pointers, and files, while it does have packages and user-defined exception handling, features not in Pascal.

The adaptation of SDVS to handle subsets of Ada included writing the formal specifications and implementations of Ada-to-state-delta translators. The items involved in this task included defining the grammar of the Ada subsets, formally specifying the two Ada-to-state-delta translator phases for each subset, building the lexical analyzer and parser for each subset, completing Common Lisp implementations of the two phases of each translator specification, testing the translator implementations, and implementing enhancements to SDVS that allow correctness proofs of Ada programs. We have experimented with the Ada translators and SDVS system modifications by proving the correctness of several small Ada programs. This involved constructing the Ada programs and their specifications, as well as proofs that the programs satisfy the specifications.

More specifically, for each subset we defined both the concrete syntax and the abstract syntax. The parser, using the concrete syntax, generates the abstract syntax for the back end (phases 1 and 2) of the translation. We used the abstract syntax to specify formally the back end of the Ada translator. This specification gives the denotational semantics of the Ada subset in terms of the state delta logic. The entire specification is written in our language DL so that it can be input to DENOTE. The Common Lisp implementation of the translator is automatically generated by DENOTE.

Whereas the previous emphasis has been on theory and tool implementation, this year's emphasis was on applications. More work is needed to enhance SDVS for sizable applications, and in 1992 we conducted two such application efforts to develop further the underlying theory and implementation.

We have been investigating the possibility of verifying pieces of Ada programs being developed for DoD Programs. In particular, we have devised a plan for coordinating efforts at The Aerospace Corporation and the Johns Hopkins University Applied Physics Laboratory (APL) to verify portions of the Midcourse Space Experiment (MSX) software using SDVS. This effort is a "shadow project" (i.e., it will not affect the MSX deliverables schedule), and it will be directed toward stressing SDVS and formulating further research strategies.

The software for the MSX Program satisfies many of the requirements that Aerospace

originally defined for an Ada application. The most important reasons for selecting the MSX software are as follows:

- Complete, accurate documentation is available.
- Access to the developers will be provided.
- The correct functioning of the software is critical to the mission's success.
- The software is not classified or restricted from access.
- There is the potential of verifying a combination of Ada software and embedded 1750A programs.
- Combining efforts of Aerospace and APL will result in a larger verification experiment than would be achievable by either team on its own.

As part of the verification process, a formal specification of the software must be created from the informal documentation supplied by the software developers. APL has already supplied Aerospace with some documentation. The documentation appears extensive, but even the best documentation has ambiguities and omissions. Hence, frequent access to the developers is necessary.

The MSX Program has a large amount of code from which to choose our example. Nevertheless, given the current stage of SDVS development, most attempts at verifying software modules will require further development of SDVS. Aerospace and APL have discussed how portions of the MSX software can be verified, given the current capabilities of SDVS. Some of the Ada constructs with which SDVS cannot currently deal will have to be rewritten (e.g. replace with procedure calls those parts that invoke tasks); the modified code will not be used in the actual MSX Program. This work will enable the team to verify some properties of the programs, while providing data to drive further work on SDVS.

The following are the main objectives of the MSX application:

- stress-test SDVS and gain more experience using SDVS to verify systems;
- identify weak and missing features in the tools and underlying theory;
- prioritize tasks for further research and development, and start to investigate the higher priority tasks;
- provide results of the verification effort to the MSX Program;
- provide another example for demonstration and publication;
- evaluate the value added from applying formal methods; and
- acquire more information about what designers need in order to apply formal methods.

A longer term objective is to integrate verification into the design process and other development tools. The MSX application may provide some useful data for this endeavor.

We have completed the first phase of the MSX verification project, which consisted of selecting and analyzing a portion of the MSX Ada code to be verified; examining the documentation pertaining to that portion; and delineating the Ada constructs that appear in that portion but that the SDVS Ada translator does not currently handle. Thus far, the prospects appear to be good for applying SDVS to a portion of the MSX software.

The second application involved selecting two sorting procedures, a bubble sort and a heap sort, from a large DoD application and verifying their correctness. For these examples we extended the Ada capability; previously, SDVS 10 had `WHILE` loops in Stage 3 Ada, but it did not have `FOR` loops. In 1992, we implemented the syntax and both translation phases for Ada `FOR` loops; with minor modifications, this implementation serves for Stage 2 VHDL as well (see Section 3). The verification of the bubble sort is complete and documented, and we are nearing the completion of the heap sort. The proofs demonstate new features of SDVS, and they are interesting because of the techniques used, the bugs they uncovered in SDVS, and their implications for possible improvements in SDVS.

In 1992 we implemented write and read facilities for **adalemmas**. Now the user can write a proved or unproved **adalemma** to a file, then read it in at some later time for invocation. If the **adalemma** is not proved, a note to that effect will accompany the proof in which that **adalemma** is used. Note that we already had the capability to read and write general lemmas; the newly implemented facility supports the relatively new modular verification technique that we refer to as "offline characterization."

See Section 4 for (1) a description of extensions and experiments being performed to prove safety properties of Ada programs, and (2) a description of a study underway to determine how Ada access types (pointers) might be formalized in SDVS.

2.2 Technical Reports

Reference [9] describes an SDVS correctness proof for a portion of operational code. This code implements a minor variant of the familiar bubble-sort algorithm, and uses `FOR`-loops and the `RECORD` structure, Ada features that are either new with this version of the SDVS translator or not previously exercised extensively. We discuss potential improvements and enhancements for SDVS, and discuss some data security problems and the ability of SDVS to treat them.

The portion of the MSX tracking-processor software that we have selected for verification in SDVS is described in [10]. We enumerate the Ada constructs that appear in this part of the software and that are not currently handled by the SDVS Ada translator, but that we intend to implement. We also mention some of the problems that we expect to encounter in the course of the project.

Reference [13] examines some of the issues that arise when SDVS is used to analyze code fragments extracted from larger bodies of production code. The code fragment must be

isolated from the larger body of code (by narrowing its interface to other program components), and it often must be altered as well (to satisfy the narrowed interface semantics or to match available SDVS capabilities). These issues are illustrated by means of an example involving a heapsort written in Ada.

3 Hardware Verification Progress – VHDL

3.1 Research and Development

Prior to 1987 we adapted SDVS to handle a subset of the hardware description language ISPS. However, ISPS has serious limitations regarding the specification of hardware at levels other than the register transfer level. In 1988 we studied some of the hardware verification research being done outside Aerospace and investigated VHDL, a DoD and IEEE standard hardware description language that was released in December, 1987. We selected VHDL as a medium for hardware description within SDVS.

From 1989 to 1991 we adapted SDVS to handle two increasingly larger subsets of VHDL, called Core VHDL and Stage 1 VHDL. Core VHDL captures many of the essential features of VHDL and is reasonably expressive; both combinational and sequential circuits can be described in this subset. Core VHDL has design entities (entity declarations and architecture bodies), declarations (constants, variables, signals, and ports), sequential statements (variable assignments, signal assignments, conditionals, `NULL` statements, and restricted `WAIT` statements with inertial delay), and the concurrent `PROCESS` statement. Stage 1 VHDL adds loops and the *unrestricted* `WAIT` statement with either inertial or transport delay. Furthermore, the treatments of processes and signals in Core VHDL were changed in Stage 1 VHDL, making the proof process more tractable and the specifications more readable. This included sequentializing processes and giving the user control over stepping through the sequential statements within a process. The dynamic flow of process execution now precisely reflects the simulation semantics of VHDL (as defined in the *VHDL Language Reference Manual* [18]).

A prerequisite to adapting SDVS to handle VHDL is to define VHDL semantics formally in terms of SDVS's underlying logic and implement a translator from VHDL to the state delta logic. The translator implementation technique for VHDL is analogous to that of the Ada effort described above. We have defined both the concrete syntax and the abstract syntax for each VHDL subset, and have implemented a parser for the language. The back end of the translator has two phases. The first phase does static semantic analysis. It performs various kinds of error checking (e.g. type checking) and collects an environment that associates with their attributes all names declared in the subject VHDL description. The second phase generates the state delta formulas. This phase receives the environment from the first phase, then uses it to translate the VHDL description incrementally into state deltas as the description is symbolically executed in the course of its correctness proof. Most of the back end is specified in a denotational manner; the sequential constructs in our subsets of VHDL are given a formal denotational semantics, while the operational semantics of the concurrent VHDL `PROCESS` construct is defined in terms of the translator's algorithm for defining the next set of valid state deltas during symbolic execution. The denotational part of the translator specification is written in the language DL so that it can function as input to DENOTE; that part of the translator specified in DL was automatically generated by DENOTE.

The Stage 1 VHDL capability has been illustrated by a sequence of examples. These include

the following:

1. a one-bit full adder, dataflow description
2. a one-bit full adder, behavioral (algorithmic) description
3. a device that switches the values of two signals
4. an XOR gate
5. a multiplexer
6. a counter
7. a sequential adder
8. a CPU-channel interface
9. a multiplier by successive addition
10. a multiplier by shifting and addition (integer implementation)
11. a multiplier by shifting and addition (bitvector implementation)

The last two examples resulted from our collaborative effort with Professor Richard Auletta of George Mason University while he was on a Navy Fellowship at the Naval Research Laboratory.

In 1992 the semantics was defined and a translator implemented for a larger VHDL subset, called Stage 2 VHDL. The translator can handle the following additional VHDL features and translator characteristics (beyond those of Stage 1 VHDL):

- VHDL design files
  - restriction: unique entity and architecture per file
- declarative parts (for declaring objects, types, subprograms) in entity declarations
- package `STANDARD`
  - predefined types: `BOOLEAN`, `BIT`, `INTEGER`, `TIME`, `CHARACTER`, `REAL`, `STRING`, `BIT_VECTOR`
  - various units of type `TIME`: `FS`, `PS`, `NS`, `US`, `MS`, `SEC`, `MIN`, `HR`
- array type declarations
  - arrays of arbitrary element type
  - bidirectional arrays, unconstrained arrays
- enumeration types

- signals of arbitrary array and enumeration types
- subprograms (procedures and functions)
  - restriction: excluding parameters of object class `SIGNAL`
  - symbolic execution of subprogram calls
- user-defined packages
- `USE` clauses for accessing packages
- concurrent signal assignment statements
  - conditional signal assignments
  - selected signal assignments
- `FOR` loops
  - loop parameter may be of either integer or enumeration type
  - loop parameter constant within loop body
  - loop bounds fixed prior to loop entry, cannot be altered by loop body
- octal and hexadecimal representations of bitstrings
- general expressions of type `TIME` in `AFTER` clauses
- ports of default object class `SIGNAL`
- statically and dynamically uniquely qualified names
  - to create new places as required by static and dynamic program structure
- interphase abstract syntax tree transformation
  - disambiguates array references and procedure calls
  - transforms concurrent signal assignment statements to `process` statements
- uniform structure (wherever possible) for Ada and VHDL translators
  - similar semantics for similar language constructs
  - both translators optimized for space- and time-efficiency

Furthermore, examples of VHDL descriptions written in Stage 1 VHDL were verified. Among those documented in 1992 are a handshake protocol for interprocess communication, the completion of the shift-and-add multiplier, a counter, and several small examples to demonstrate `TRANSPORT` delay, embedded `WAIT` statements, and `LOOP` statements.

Under an Aerospace-sponsored project, we have studied and proposed an SDVS semantics for structural constructs in VHDL hardware descriptions, such as component declarations and component instantiation statements. Once this is implemented, a hierarchical proof can be developed to mirror the hierarchical structure in the hardware description.

3.2 Technical Reports and Papers

Reference [7] documents a formal semantic specification of Stage 2 VHDL. Now implemented in SDVS, the Stage 2 VHDL translator represents the latest phase of our research toward proving properties of VHDL descriptions. The semantics is primarily specified denotationally, although the second-phase semantics of the VHDL simulation cycle has a direct operational implementation in the VHDL translator code.

In [6], we illustrate, by a sequence of examples, how SDVS can be used to create formal specifications for and correctness proofs of hardware descriptions in Stage 1 VHDL.

Reference [15] gives an overview of SDVS; it presents an introduction to the verification approach and the current facilities of SDVS for hardware verification. In particular, it describes the components of the SDVS system, the underlying model of computation, the proof procedure, and SDVS 10's capability for verifying VHDL hardware descriptions.

4 Research in Safety and Pointers

4.1 Research

4.1.1 Safety

In 1988 we completed a study of how claims of avoidance can be specified in SDVS. Examples were given of avoidance claims in SDVS by means of negations of state deltas, and an extension to the state delta syntax was proposed to allow more general claims. The concept of an "invariance condition" was added to the state delta. This extension permits us to specify a version of the `until` temporal operator. (This invariant should not be confused with the invariant used in the **induct** command. The latter will be referred to as the induction invariant in our reports.)

In 1989 we investigated enhancements to SDVS software to extend the state delta logic. We determined how the implementation of existing proof rules should be modified, and we also defined some new proof rules for reasoning about state deltas that have invariance conditions.

In 1990 we implemented and demonstrated these enhancements. We developed an example proof to illustrate the strength of the invariance extension of SDVS. The example involves proving a safety property of a concurrent program. The state delta translation of the concurrent program and the proof of the safety property are not possible in SDVS without invariance. Versions 9, 10, and 11 of SDVS allow a flag to be set to enable/disable this new capability; if it is disabled, SDVS assumes the invariants of all state deltas are TRUE, and the system behaves as if the state deltas did not have the invariant field.

In 1991 we started to design applications and specify their safety properties using the extended state delta logic, and to demonstrate the verification of these applications. In particular, we continued work on the verification of a safety property of a simple concurrent program. This proof requires a new SDVS command, **omegainduct**.

In 1992 we implemented the **omegainduct** command, which enables us to prove safety properties of nonterminating programs. This command and an extension of the **negate** command restrict the logic of SDVS to a logic that is true only of timelines that are either finite or isomorphic to the natural numbers. Our work on this command is a natural extension of our previous work on invariance. Using this command, we have proved, in SDVS 11, a state delta that was not provable in SDVS 10.

Also in 1992, we experimented with alternative Ada translations in order to prove safety properties about Ada programs. We implemented a new flag, **weaknext_tr**, that adds a strong invariant field to the state deltas that are generated by the SDVS language translators: specifically, it adds the formula $\#all = .all$ to the invariant field of those state deltas. This addition and a change to the **apply** command result in a different model of the symbolic execution of programs in SDVS. In this model, an execution of a simple statement of a program is interpreted as a *discrete* state transition; roughly, there are no states between the precondition and postcondition states. We have used these extensions of SDVS in a proof

of a safety property of a terminating Ada program and in another proof of a safety property of a nonterminating Ada program. These are the first SDVS proofs of safety properties of Ada programs.

4.1.2 Ada Access Types (Pointers)

While the verification of programs involving pointers has been troublesome in some verification systems, we believe, after our study in 1992, that such proofs are feasible in SDVS. In particular, we developed a method for handling Ada access types in SDVS, i.e., a method that allows SDVS to translate and reason about Ada programs containing access types. The method is built upon "higher-order places," i.e., places that have other places as contents.

Briefly, using the machine model, a regular (nonpointer) variable x corresponds to a memory location (place). Its contents are a certain string of bits. In the state delta language the "current" contents of x (before a state change) are designated by $.x$, and the contents after a state change by $\#x$. However, a pointer variable z does not correspond itself to a memory location, but simply to a list of potential memory locations. The contents of z ($.z$ or $\#z$) would be one of those potential locations. Thus, the contents of that location would be a certain string of bits; $.(.z)$ denotes the *current* contents of the location *currently* pointed to by z before a state change, $.(\#z)$ denotes the *old* contents of the *new* location pointed to by z after a given state change, $\#(\#z)$ denotes the *new* contents of the *new* location pointed to by z after a given state change, and $\#(.z)$ denotes the *new* contents after a state change of the *old* location pointed to by z. The place and covering solvers of SDVS currently handle such constructs correctly for the most part. For example, $.(.z)$, $\#(\#z)$, and $\#(.z)$ are handled correctly, while the treatment of $.(\#z)$ needs to be improved.

4.2 Technical Reports

In [4] we introduce the recently implemented **omegainduct** command. We discuss its theory and implementation and illustrate its use in a proof of a safety property of a simple concurrent program. We note that this command is valid only for timelines in which $\omega + 1$ cannot be embedded.

Reference [5] describes recent enhancements to the implementation of invariance in SDVS, the purpose of the new **weaknext_tr** flag, and enhancements to the **omegainduct** command. We use these enhancements to prove a safety property of a terminating Ada program and a safety property of a nonterminating Ada program.

In [12] we discuss how proofs of typical safety properties of programs in temporal-logic-based systems can be facilitated by the use of two proof rules: the Rule of Negation and the ω-Induction Rule. We show that each of these rules is (independently) valid only on timelines of certain order types; the joint use of the two rules is valid only on timelines that are finite or ordered like the natural numbers. We demonstrate the use of these rules in SDVS by giving proofs of two safety properties of a simple concurrent program.

In [8] we propose a method for handling Ada access types in SDVS. We give the state delta semantics for various Ada constructs involving access types, discuss the theory and implementation of higher-order places, and give an example of an Ada program involving access types.

5 General System Development

5.1 Development

We have completed the Sun 4 version of SDVS 11 (along with a description of the installation procedure and an enlarged test suite) and the *SDVS 11 Users' Manual* [11]. SDVS 11 currently runs under both Franz Allegro Common Lisp (FACL) 4.1 and Lucid Common Lisp (Lucid) 4.0.[2] The manual has been updated to reflect the new subsets of Ada and VHDL. All examples in the manual have been executed using SDVS 11 on our Solbourne computer (which is binary compatible with a Sun 4). In addition, we have updated the tutorial [14] to reflect changes made to version 11.

SDVS has been released to the following organizations: California Institute of Technology, George Mason University, Johns Hopkins University Applied Physics Laboratory, MITRE Corporation, Naval Research Laboratory, National Security Agency, Rome Laboratory, Trusted Information Systems, and University of California at Santa Barbara. Aerospace has purchased a "runtime generator" license agreement from Franz, Inc., which allows Aerospace to deliver SDVS to its end users without requiring them to purchase a Common Lisp system (the runtime version will remove some features from the development version of Common Lisp).

FACL has Common Windows, an interface to the X-windows system. A facility was added to SDVS allowing "remote demonstrations" of SDVS by using Common Windows. This facility allows Aerospace to give a demonstration of the capabilities of SDVS to users at any site on the Internet who are running the X-windows system. When the remote demonstration facility is invoked, a window will appear on a remote display containing a trace of the local activity of an SDVS session. Control of the SDVS session remains solely with the local Aerospace user; i.e., the remote user has no input capability to the SDVS session.

Under an Air Force sponsored project, we are exploring the potential for expanding the SDVS user interface to make use of bit-mapped displays and windowing technologies. Specifically, we are using X-windows as our windowing platform. X-windows is quickly becoming a portable (de facto) standard windowing system. We have started to build the interface using Common Windows. A version of the Ada program trace facility has been completed and fully integrated into the SDVS verification system. As an Ada program executes symbolically, the program statements corresponding to the execution are highlighted in an X-window, allowing the user to follow the symbolic execution more easily. A "scroll bar" is available to move the program in an X-window when the entire program does not fit in the window. Scrolling preserves the highlighting. The trace facility attempts to present as much of the current statement as possible by scrolling automatically when that statement either is not on the screen or does not fully fit on the screen. This facility has been expanded to include offline characterization as implemented in the **createadalemma** and **proveadalemma** SDVS commands. This expansion will allow multiple symbolic-execution trace windows to handle proofs of subroutines (through offline characterization) of Ada

[2] Aerospace is awaiting the release of Lucid Common Lisp version 4.1; SDVS 11 will be released under that system when it is available.

programs. We are currently examining other interface building tools (e.g. Common Lisp Interface Manager, or CLIM, from Franz Inc.).

5.2 Technical Reports

Reference [11] is the users' manual for SDVS 11. Although it is primarily a reference manual, it has tutorial aspects as well. The manual contains descriptions of the following:

- underlying logic (state delta logic)
- proof language
- user interface
- ISPS verification capability
- Ada verification capability
- VHDL verification capability
- domains defined in the SDVS Simplifier and capabilities of static solvers

All facets of the system are illustrated with example SDVS sessions.

Reference [14] is a tutorial for SDVS. This tutorial contains a description of most, but not all, of the proof capabilities of SDVS. (The *SDVS 11 Users' Manual* should be consulted for a more comprehensive account.) The description is embedded in numerous examples of proofs in SDVS. In particular, the tutorial contains descriptions and examples of the following:

- state delta logic
- dynamic and static proof commands
- some SDVS data types
- quantification
- techniques that are available for verifying hardware descriptions and programs written in VHDL, Ada, or ISPS

6 Conclusion

Much progress was made in fiscal year 1992 in extending SDVS for verifying software and hardware with respect to a wide range of specifications. The translator implementation technique that we developed in 1987 has continued to be very successful, and it has been used this year for the translator implementations of Stage 3 Ada and Stage 2 VHDL. Our advances since 1987 have enabled us to define the semantics of new, more advanced language features of Ada and VHDL in terms of the state delta logic, and we are in the process of using SDVS to verify interesting examples written in Ada and VHDL. The ISPS verification facility remains a feature of our system.

This year's work has contributed significantly to our goal of using SDVS to verify computer systems from high-level software to hardware. Next year we will continue to advance research in the areas of semantics, temporal logic, and theorem provers, in order to make verification possible for critical computer systems. In addition to extending our capabilities to verify programs and hardware descriptions written in larger subsets of Ada and VHDL, our plans include developing a wide range of verification examples to illustrate our capabilities, developing techniques to make the proof process more efficient and manageable, improving and experimenting with the user interface, and permitting more user-defined extensions.

We have distributed the system and are providing user support to a number of new sites. The expanding SDVS user base is providing us important feedback from users. We expect that this activity will increase, and we look forward to seeing that SDVS is applied as widely as possible. This has two principal benefits: first is the direct, anticipated benefit of improving the quality of critical hardware and software; and second is the valuable feedback from real-world application that will enable us to continue to build a system that is responsive to users' problems.

References

[1] J. V. Cook, "Verification of the C/30 Microcode Using the State Delta Verification System (SDVS)," in *Proceedings of the 13th National Computer Security Conference*, (Washington, D. C.), pp. 20–31, National Institute of Standards and Technology/National Computer Security Center, October 1990.

[2] B. H. Levy, "An Overview of Hardware Verification Using the State Delta Verification System (SDVS)," in *Proceedings of the 1991 International Workshop on Formal Methods in VLSI Design*, (Miami, Fla.), ACM, January 1991.

[3] J. V. Cook, "The Language for DENOTE (Denotational Semantics Translation Environment)," Technical Report TR-0090(5920-07)-2, The Aerospace Corporation, September 1990.

[4] T. Menas, "Safety, Invariance, and a New Induction Command in SDVS," Technical Report ATR-92(2778)-1, The Aerospace Corporation, September 1992.

[5] T. Menas, "Safety Properties of Terminating and Nonterminating Ada Programs in SDVS," Technical Report ATR-92(2778)-2, The Aerospace Corporation, September 1992.

[6] I. V. Filippenko, J. M. Bouler, and B. H. Levy, "Some Examples of Verifying Stage 1 VHDL Hardware Descriptions Using the State Delta Verification System (SDVS)," Technical Report ATR-92(2778)-3, The Aerospace Corporation, September 1992.

[7] I. V. Filippenko, "A Formal Description of the Incremental Translation of Stage 2 VHDL into State Deltas in the State Delta Verification System (SDVS)," Technical Report ATR-92(2778)-4, The Aerospace Corporation, September 1992.

[8] L. G. Marcus, "The Semantics of Ada Access Types (Pointers) in SDVS," Technical Report ATR-92(2778)-5, The Aerospace Corporation, September 1992.

[9] J. Doner, "SDVS Verification of a Stage 3 Ada Program," Technical Report ATR-92(2778)-6, The Aerospace Corporation, September 1992.

[10] T. Menas, "A Proposal for the Verification in SDVS of a Portion of the MSX Tracking Processor Software," Technical Report ATR-92(2778)-7, The Aerospace Corporation, September 1992.

[11] L. G. Marcus, "SDVS 11 Users' Manual," Technical Report ATR-92(2778)-8, The Aerospace Corporation, September 1992.

[12] T. K. Menas and L. G. Marcus, "Timelines and Proofs of Safety Properties in the State Delta Verification System (SDVS)," Technical Report ATR-92(2778)-9, The Aerospace Corporation, September 1992. Submitted to *Journal of Automated Reasoning.*

[13] L. A. Campbell, "Isolating and Transforming an Ada Heapsort Program for SDVS Analysis," Technical Report ATR-92(2778)-11, The Aerospace Corporation, September 1992.

[14] T. K. Menas, "SDVS 11 Tutorial," Technical Report ATR-92(2778)-12, The Aerospace Corporation, September 1992.

[15] B. Levy, I. Filippenko, L. Marcus, and T. Menas, "Using the State Delta Verification System (SDVS) for Hardware Verification," in *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience: Nijmegen, The Netherlands* (ed. V. Stavridou, T. F. Melham, and R. T. Boute), pp. 337–360, North-Holland, June 1992.

[16] I. V. Filippenko and L. G. Marcus, "Integrating Structural VHDL Hardware Descriptions into the State Delta Verification System (SDVS)," Technical Report ATR-92(8180)-1, The Aerospace Corporation, September 1992.

[17] J. V. Cook, I. V. Filippenko, B. H. Levy, L. G. Marcus, and T. K. Menas, "Formal Computer Verification in the State Delta Verification System (SDVS)," in *Proceedings of the AIAA Computing in Aerospace Conference*, (Baltimore, Maryland), pp. 77–87, American Institute of Aeronautics and Astronautics, October 1991.

[18] IEEE, *Standard VHDL Language Reference Manual*, 1988. IEEE Std. 1076-1987.